ACTA DIDACTICA NAPOCENSIA
Volume 10, Number 3, 2017

IMAGE PROCESSING ALGORITHMS IN THE SECONDARY SCHOOL PROGRAMMING EDUCATION

István Gerják

Abstract: Learning computer programming for students of the age of 14-18 is difficult and requires endurance and engagement. Being familiar with the syntax of a computer language and writing programs in it are challenges for youngsters, not to mention that understanding algorithms is also a big challenge. To help students in the learning process, teachers need to increase their engagement and motivation by giving them interesting and entertaining tasks. Implementing different image processing algoritms can be very efficient for enhancing their programming skills, since by running their programs on real pictures taken by cameras, they can examine the effects of these programs. This article carefully selects and introduces image processing algorithms that are based on basic computer algorithms and are understandable for secondary school students. It also shows how to apply them in programming classes and how to provide the discovery learning for students.

Key words: computer programming, digital image processing, algorithms

1. Introduction

During digital image analysis, the goal is to analyse the picture, understand and correctly interpret the visual information in it and finally recognise what it depicts. To achieve this, transformations are needed to be performed, that result in an image with better properties than the original one. These transformations are usually algorithms, most on strong mathematical basis. That is one of the reasons, why it is not taught in secondary schools. The second reason is that neither the final exam nor the study contests require the knowledge of digital images processing. Students need to solve drawing and image editing tasks that can be performed by using image editor programs such as Paint or Gimp, or sometimes PowerPoint is also suitable. They get source images to construct the sample image and also, need to change colours, to add texts or remove parts from the source images.

Naturally, for more talented students we can teach image processing that will enable them to perform research at their early stage of higher education and hence opening new application for digital image processing [1]. We can also teach image processing algorithms related to the topics of final exam of informatics. For example, students need to know compressing and the different compressed digital image formats demonstrated the way of teaching JPEG coding for secondary school students [4]. A digital image processing course can provide the perfect combination of the theory and experiment teaching, the practice and innovation training [8, 9].

If our goal is to enhance our students' programming skills and make the programming lessons more enjoyable, coding some image processing algorithms is a very useful option. They like dealing with digital images, and coding algorithms that change the original image is more interesting than processing text data after reading them from a text file.

Hungarian students, who desire to pass the advanced level final exam of informatics, need to know how to write computer programs that can read from a file and display the processed data to both files and monitor in a high level programming language, such as C#, Java or C++. They also need to know the basic computer algorithms (counting, searching, sorting, minimum/maximum finding, etc.) and stuctures (array, matrix, record, etc.)

Students sometimes find it hard to learn a programming language. They often confront unfamiliar programming terms and are required to visualize the processes that happen in computer memory.

Received July 2017.

Others find this a burden and end up memorizing the processes without understanding them. This situation invariably leads students to get low grades in their programming subjects and abandon learning programming. Their motivation and interest can be increased if the practical exercises are applicable to real life to feel the usefulness of their activity. [3, 6, 7]

In this paper, we introduce some of the image processing algorithms that can be used in programming lessons and the way of implementing them to be understandable for secondary school students. Instead of reading numbers and letters from the standard console, these programs read the pixel values from real image files and after completing the image processing and operation the result is another image file. We use C# codes but similar methods are available in Java and C++ as well.

Students can use their own images and test their programs using the photos taken by their mobiles or find other images via Internet. These algorithms can be implemented in different ways to give the chance to improve our students' programming skills.

2. Getting the pixels of an image and changing them

A digital image can be considered a matrix in which elements store the colour of a point in the image. The key point is how to get the pixel values (colours) of an image. In C#, we need to instantiate the Bitmap class and open the file:

```
Bitmap sourceimage= new Bitmap(”myimage.jpg”);
```

Without giving the full path, the image file must be stored in the folder of the application. To get the RGB code of the pixels, GetPixel method must be used. Since the RGB code contains three integer numbers, for storing the values a Color variable is necessary to store. The Width and the Height properties gives the size of the image. Figure 1 shows the code that saves the colour values of an image in a matrix.

```
Bitmap sourceimage= new Bitmap("myimage.jpg");
Color[,] image = new Color[sourceimage.Height, sourceimage.Width];
for (int x = 0; x < sourceimage.Width; x++)
{
   for (int y = 0; y < sourceimage.Height; y++)
   {
      image[y,x] = sourceimage.GetPixel(x, y);
   }
}
```

Figure 1. *Getting the pixels' RGB codes of an image in C#*

Naturally, the image matrix can be used for general programming tasks such as find a given RGB value in the image, find the darkest RGB code of the image, etc. However, more interesting and more spectacular tasks can be presented in the area of digital image processing. It is very common to obtain an image in black and white. To achive this, the first step is to construct the greyscale version of the image. Figure 2 shows the C# code of this transform:

```
Bitmap sourceimage= new Bitmap("myimage.jpg");
Bitmap targetimage =new Bitmap(sourceimage.Width, sourceimage.Height);
Color[,] image = new Color[sourceimage.Height, sourceimage.Width];
for (int x = 0; x < sourceimage.Width; x++)
{
 for (int y = 0; y < sourceimage.Height; y++)
 {
  Color pxl = sourceimage.GetPixel(x, y);
  int grayScale = (int)((pxl.R * 0.3) + (pxl.G * 0.59) + (pxl.B * 0.11));
  Color nc = Color.FromArgb(pxl.A, grayScale, grayScale, grayScale);
  targetimage.SetPixel(x, y, nc);
 }
}
targetimage.Save("greyscale.jpg");
```

Figure 2. *Converting an image to greyscale*

To get the greyscale code, averaging the RGB codes is not correct but it is worth trying it and comparing the result with the correct image [2]. To save the greyscale image we need another Bitmap object, targetimage. 'FromArgb' method constructs a Color object from the alpha value (opaque level) and the three colour codes. The 'SetPixel' method sets the pixel values of the target image. At the end, the program saves the new image as a file in the folder of the application.

If we want to show both the original and result images in our windows form application just add two pictureboxes to our application and load them. Figure 3 shows the new code:

```
pictureBox1.Load("myimage.jpg");
Bitmap sourceimage= new Bitmap("myimage.jpg");
Bitmap targetimage =new Bitmap(sourceimage.Width, sourceimage.Height);
Color[,] image = new Color[sourceimage.Height, sourceimage.Width];
for (int x = 0; x < sourceimage.Width; x++)
{
 for (int y = 0; y < sourceimage.Height; y++)
 {
  Color pxl = sourceimage.GetPixel(x, y);
  int grayScale = (int)((pxl.R * 0.3) + (pxl.G * 0.59) + (pxl.B * 0.11));
  Color nc = Color.FromArgb(pxl.A, grayScale, grayScale, grayScale);
  targetimage.SetPixel(x, y, nc);
 }
}
targetimage.Save("greyscale.jpg");
pictureBox2.Load("greyscale.jpg");
```

Figure 3. *Displaying the new image in the form*

Using this code, it is easy to invert images or get the binary versions (black and white) of them. To invert and image, the RGB codes must be substracted from 255:

```
targetimage.SetPixel(x, y, Color.FromArgb(255 - pxl.R, 255 - pxl.G, 255 - pxl.B));
```

To get the black and white version of an image, we need to find a threshold number to decide whether the pixel will be black (0) or white (1). Figure 4 shows the new loop body:

```
Color pxl = sourceimage.GetPixel(x, y);
int grayScale = (int)((pxl.R * 0.3) + (pxl.G * 0.59) + (pxl.B * 0.11));
if (grayScale < 120)
   targetimage.SetPixel(x, y, Color.FromArgb(0, 0, 0));
else
   targetimage.SetPixel(x, y, Color.FromArgb(255, 255, 255));
Color nc = Color.FromArgb(pxl.A, grayScale, grayScale, grayScale);
targetimage.SetPixel(x, y, nc); image[y, x] = sourceimage.GetPixel(x, y);
```

Figure 4. *Binarization of an image*

Why is it 120? Ask our students to change it and try other threshold numbers for other images. There are several methods to find the proper threshold number for a given image, for example the Otsu‘s or the Niblack’s methods. For different images, let our student find the best one or we can implement for instance the Otsu’s method in our application as well.

3. Calculating the histogram of an image

A very common programming task is to define how many there are of different items from a series. To prepare our students for this type of tasks, show them the histogram of images. Modern cameras display the histogram of the current image since it can us tell us whether our image has been properly exposed, whether the lighting is harsh or flat, and what adjustments will work best. A RGB histogram is created when the camera scans through each of these RGB brightness values and counts how many there are at each level from 0 through 255 [2]. Examining the histogram of an image we can determine whether the intensity values **spread** (good) out or **bunched up** (bad), so it is **underexposed**, **overexposed** (see Figure 5) or properly exposed. The contrast of a grayscale image indicates how easily objects in the image can be distinguished. **High contrast** image: many distinct intensity values, there are almost no midtones. It is either very dark (sky) or very bright (marble tiles). **Low contrast**: image uses few intensity values, so there are no shadows and no highlights.

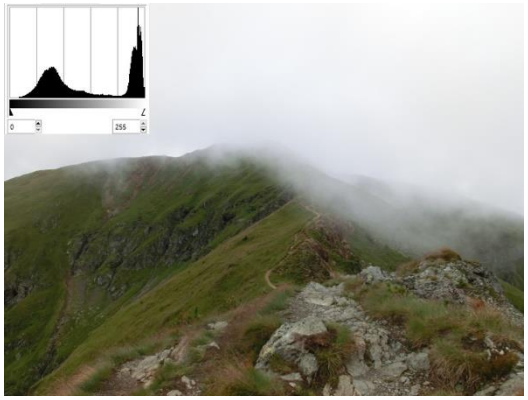

Figure 5. *An author’s overexposed photo with its histogram*

The following code determines the histogram of an image after calculating the grey scale values (Figure 6).

```
Color pxl;
int size = sourceimage.Width * sourceimage.Height;
for (int x = 0; x < sourceimage.Width; x++)
   for (int y = 0; y < sourceimage.Height; y++)
   {
      pxl = sourceimage.GetPixel(x, y);
      int temp = (int)((pxl.R * 0.3) + (pxl.G * 0.59) + (pxl.B * 0.11));
      myHistrogram[temp]++;

   }
```

Figure 6. *Determining the histogram of an image*

Ask our students to draw the histogram of the image from the ‘myHistogram’ array on the form. Naturally, the occurrence values must be normalized before drawing the histogram. It is also possible to write a program that can categorize the different pictures. For example, we have 50 picures and select the underexposed ones.

4. 2D discrete convolution

In digital image processing, the two dimensional discrete convolution is used. Formula 1 shows the formula of this operation:

$$newimage[x,y] = \sum_{i=0}^{y-1} \sum_{j=0}^{x-1} image[i,j] \bullet psf[x-i, y-j]$$

Formula 1. *2D discrete convolution*

PSF (*Point Spread Function*) is usually a small matrix (typically 3x3 or 5x5) [2]. From a programming point of view, 2D convolution is two for loops in which the parts of the image are covered by the PSF and the values below each other must be multiplied and their product must be added. The result value will be new value of the middle point of the new image. It is worth showing real examples for students to understand the algorithm before coding it [5]. In the following example, the PSF is 3x3 with 9 (or 1/9) values and let us suppose that in the current step it covers the part of the image denoted by bold letters and borders (see Figure 7).

Image

11	10	12	11	12
10	**13**	**15**	**20**	34
12	**21**	**11**	**12**	13
12	**14**	**15**	**16**	16
18	11	10	12	12

PSF

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

13•1/9+15•1/9+20•1/9+21•1/9+11•1/9+12•1/9+14•1/9+15•1/9+16•1/9 = 137/19 ≈ 7

Figure 7. *Example of 2D discrete convolution for image*

In this step, after multiplying the numbers and adding them, 11 will be replaced by 7 in the new image. If we adjust the PSF to the left upper corner of the image, the first pixel that will be changed is in the second row and the second coloumn. (13 is in the example). If we want to apply the convolution for the pixels in the corners and on the edges of the image, we need to add new rows and coloumns to the image repeating the edges on the opposite sides as the nex code segment shows. First, it is necessary to copy the RGB codes of the image to the matrix 'a[n,m]', so its elements are record type containing 3 integer numbers for the codes (see Figure 8).

```
for (int i = 0; i < n + 1; i++)
{
   a[i, 0].r = a[i, m + 1].r; a[i, 0].g = a[i, m + 1].g; a[i, 0].b = a[i, m + 1].b;
   a[i, m].r = a[i, 1].r; a[i, m].g = a[i, 1].g; a[i, m].b = a[i, 1].b;
}

for (int i = 0; i < m + 1; i++)
{
   a[0, i].r = a[n + 1, i].r; a[0, i].g = a[n + 1, i].g; a[0, i].b = a[n + 1, i].b;
   a[n, i].r = a[1, i].r; a[n, i].g = a[1, i].g; a[n, i].b = a[1, i].b;
}
```

Figure 8. *Adding new coloumns and rows to the image*

The following figure (Figure 9) represents the convolution code supposing a PSF of 3x3 size. Since the operation can result in values above 255 and below 0, it is necessary to replace the incorrect values to either 0 or 255.

```
int sumr, sumg, sumb = 0;
for (int i = 0; i < n - 3 / 2; i++)
{
   for (int j = 0; j < m - 3 / 2; j++)
   {
      sumr = 0; sumg = 0; sumb = 0;
      for (int k = 0; k < 3; k++)
         for (int l = 0; l < 3; l++)
         {
            sumr += a[i + k, j + l].r * psf[k, l];
            sumg += a[i + k, j + l].g * psf[k, l];
            sumb += a[i + k, j + l].b * psf[k, l];
         }
      sumr = sumr / divide > 255 ? 255 : sumr / divide; sumr = sumr < 0 ? 0 : sumr;
      sumg = sumg / divide > 255 ? 255 : sumg / divide; sumg = sumg < 0 ? 0 : sumg;
      sumb = sumb / divide > 255 ? 255 : sumb / divide; sumb = sumb < 0 ? 0 : sumb;
      alias[i + 3 / 2, j + 3 / 2].r = sumr;
      alias[i + 3 / 2, j + 3 / 2].g = sumg;
      alias[i + 3 / 2, j + 3 / 2].b = sumb;
   }
}
```

Figure 9. *The code of the 2D disceret convolution in C#*

In this code, the alias matrix contains RGB codes of the new image after performing the convolution with a given PSF. The last step is to create the new image from this matrix calling the SetPixel method of the 'targetimage' object in two loops like it is showed in Figure 2.

Applying 2D convolution in the programming classes gives a chance to the exploratory learning. Students can use different PSF-s for edge detecting and for blurring. Using the proper, PSFs they can adjust the sharpness of their images.

To make this operation clearer for them, instead of real images they can perform it on numbers and the result will be a new matix of numbers as Kiraly showed (Kiraly, 2016.)

5. Basic morphological algorithms

Image segmentation is the process of dividing an image into multiple parts. This is mainly used to identify objects or other relevant information in digital images. One way to simplify the problem is to change the grayscale image into a binary image (black and white image), and then perform a morphological operator to eliminate the possible errors of binarization.

Morphological operators often take a binary image and a structuring element as input and combine them using a set operator (intersection, union, inclusion, complement). Generally, the structuring element is sized 3×3 and has its origin at the center pixel [2].

The simplest operators are the dilation and the erosion, and their combinations: opening and closing. For dilation, the structuring element is shifted over the image and at each pixel of the image, its elements are compared with the set of the underlying pixels. If there is **at least one** 0 in it, the center pixel will be changed for 0, if there is not, it will be changed for 1 (or 255, which is black in the RGB system).

For erosion, the method is almost the same, but if **all the pixels** in the covered matrix segment of 3x3 are 0, the center pixel will be changed for 0, otherwise it becomes 1 (or 255 in the implementation). Hence, dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries. Operation opening is defined as an erosion followed by a dilation using the same structuring element for both operations. Closing is opening performed in reverse order. It is defined as a dilation followed by an erosion using the same structuring element for both operations.

Notice, that dilation and erosion are based on a basic computer algorithm: counting. Count the number of ones in the current covered area. If the number of zeroes is 9, the center pixel will be 0 otherwise 1 for erosion. If the number of zeroes is 0, the center pixel will be 0 otherwise 1 for dilation.

Nevertheless, it is also possible to apply logical operators. Figure 10 represents an implementation of the dilation operation using logical OR operator.

```
for (int i = 1; i < n - 1; i++)
   for (int j = 1; j < m - 1; j++)
   {
      bool ok = false;
      for (int k = -1; k <= 1; k++)
         for (int l = -1; l <= 1; l++)
             ok = ok || a[i + k, j + l].r == 0 ? true : false;
      alias[i, j].r = ok ? 0 : 255;
      alias[i, j].g = ok ? 0 : 255;
      alias[i, j].b = ok ? 0 : 255;
   }
```

Figure 10. *The code of dilation applying logical OR*

The image is stored in matrix ‘a’, its elements are records containing three integer values for RGB codes. The new image is stored in matrix ‘alias’. Since fields ‘r’, ‘g’ and ‘b’ store the same number (either 0 or 1), checking any of them is sufficient.

The code of erosion is very similar, see Figure 11.

```
for (int i = 1; i < n - 1; i++)
   for (int j = 1; j < m - 1; j++)
   {
      bool ok = true;
      for (int k = -1; k <= 1; k++)
         for (int l = -1; l <= 1; l++)
            ok = ok && a[i + k, j + l].r == 0 ? true : false;
      alias[i, j].r = ok ? 0 : 255;
      alias[i, j].g = ok ? 0 : 255;
      alias[i, j].b = ok ? 0 : 255;
   }
```

Figure 11. *The code of erosion applying logical AND*

If all the numbers in the area coverd by the structuring element are 0, the center pixel will be 0, otherwise it will be 255, which is black in RGB.

Notice, that these operations can also be implemented as a set of operators (intersection and union), so they are also efficient for improving our students’ skills in this field. Coding opening and closing requires implementing erosion and dilation as procedures that can help student to understand the usefulness of them.

To shed light on the effect of these operators, ask student to find proper images and run their programs with different binarised images.

6. Conclusion

Digital image processing algorithms can be used for improving the programming skills of students extremely entertainingly and effectively since they can work with digital images and the output of their programs is always another image not a simple text file. Besides, these algorithms give the chance for our students to get an insight into a new field of computer applications: digital image processing.

In this paper, some image processing algorithms were presented that can be easily understandable for secondary school students and related to basic computer algorithms that are part of the IT curriculum.

References

[1] Al-Ghaib, H.; Adhami, R. (2012), An E-learning interactive course for teaching digital image processing at the undergraduate level in engineering, Interactive Collaborative Learning (ICL). 2012 *15th International Conference* on, vol., no., pp.1,5, 26-28 Sept. 2012 doi: 10.1109/ICL.2012.6402146

[2] Gonzales, Rafael C., and Woods, Richard E. (2008), *Digital Image Processing*, Third Edition. Pearson Education Inc., 2008.

[3] Garris, R., Ahlers, R., Driskell, J.E (2002), Games, motivation and learning, A research and practice model. *Simulation and Gaming*, 33(4), 441-467.

[4] Király, S. (2012), Demonstrating the feature of energy saving of transforms in secondary schools. *Teaching Mathematics and Computer Science*, 10/1 (2012), 43-55.

[5] Király, S. (2016), How to Implement An E-Learning Curriculum to Streamline Teaching Digital Image Processing. *Acta Didactica Napocensia*, 2016.Volume 9, Number 2, 2016

[6] P. de BYL, J. HOOPER (2013), *Key Attributes of Engagement in a Gamified Learning Environment*, In H. Carter, M. Gosper and J. Hedberg (Eds.), Electric Dreams. Proceedings ASCILITE 2013 Sydney. (pp.221-230)

[7] Pellas, N. (2014), The influence of computer self-efficacy, metacognitive self-regulation and self-esteem on student engagement in online learning programs. *Evidence from the virtual world of Second Life Computers in Human Behaviour*, 35, 157-170, 2014.

[8] Zhonghua Wang and Jun Gu (2017), Teaching and Practice Mode Reform in Digital Image Processing Curriculum, *International Journal of Information and Education Technology, Vol. 7, No. 7,* July 2017.

[9] Yigang Wang, Shengli Fan, Jialin Cui, Zhuoyuan Wang, Lingong Li (2012). Practical Teaching Reform for Digital Image Processing Based on Project-Driven. Emerging Computation and Information teChnologies for Education, *Proceeding of 2012 International Conference on Emerging Computation and Information technologies for Education* (ECICE 2012)

Author

István Gerják, Kálmán Kandó Faculty of Electrical Engineering, Óbuda University, Budapest, Hungary, e-mail: gerjakist@gmail.com